METHODS AND APPARATUS FOR MODIFYING PROGRAMS STORED IN READ ONLY MEMORY

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates to embedded systems. More particularly, the invention relates to methods and apparatus for modifying embedded system programs stored in read only memory (ROM).

2. State of the Art

Digital computing systems are typically known to include hardware and software. Software is typically stored on a writable medium such as a magnetic disk. As most computer users know, software often contains errors or mistakes which prevent it from functioning properly. Software vendors often publish updates or "patches" to correct errors in software. An update is typically a new complete version of the original software which is intended to replace the entire original program. The original software is erased from the writable medium and the replacement software is written to the writable medium in place of the original. A patch is a program which is run to modify the original software. The patch program finds the portion(s) of the software which need to be replaced and overwrites them with replacement code.

Today, digital computing systems are ubiquitous and often unseen. These systems are designed to perform specialized functions such as controlling the operation of an appliance, a motor vehicle, or a computer peripheral device such as a modem or a printer. Such computing systems are usually referred to as "embedded systems". They include a microprocessor and software which is typically stored on a read only memory (ROM). ROM is advantageous because it is non-volatile, small, inexpensive, and energy efficient. Program code stored on ROM is usually called "firmware" rather than software because once it is stored on ROM, the code cannot be modified. The code takes on the quality of hardware in the sense that in order to change it, the physical ROM device must be replaced. Indeed, many embedded systems have "socketed" ROM so that the ROM can be easily unplugged and replaced with a new ROM if the program needs to be changed. However, that is not always practical or convenient.

One known method for alteration of firmware programs in ROM-based systems is disclosed in U.S. Patent Number 4,607,332, issued on August 19, 1986 to Goldberg. The method allows for dynamic alteration of ROM programs with the aid of random access memory (RAM) and through the use of the standard linkages associated with subroutine calls in the system processor. When each ROM-based routine is written, one program statement is a call to a ROM-located processing routine which searches a RAM-located data

structure. If there exists a correspondence between information passed on the call to the processing routine and certain elements of the data structure, a RAM-based program is substituted for the ROM-based program. After adjusting the processor to the states just after the call to the ROM-based program, the processing routine effects a transfer to the replacement routine in RAM. The location of this replacement routine is also found in the data structure. The main disadvantage of the method proposed by Goldberg is that it is inefficient. It uses up too much space in ROM and RAM. It also requires an external compiler/interpreter. Since the processing function is ROM-based, it is limited in scope and potential to be modified. The ROM-based function also presents excessive processing overhead since it must perform a search for possible replacement code even when no replacement code exists

Another system for altering ROM-based firmware is disclosed in U.S. Patent Number 5,740,351, issued on April 14, 1998 to Kasten. The system relies on an "extensible interpreter", i.e. a modified FORTH kernel and a plurality of customizable call outs (CCOs). CCOs are present in the ROM-based program. When a CCO is encountered during execution of the program, the modified FORTH kernel takes control and looks for the called function or parameter. If it is found (in RAM or in ROM), it is executed or fetched and the result is returned by the modified FORTH kernel to

the next instruction in the ROM program. The system is primarily intended for interactive use with a dumb terminal during "debugging" of the ROM-based program. CCOs in RAM must be defined using the modified FORTH kernel. The main advantage of Kasten over Goldberg is that Kasten does not require an external compiler/interpreter. Disadvantages of Kasten are that it is limited to modifications made using the FORTH programming language and it is inefficient, requiring that a substantial part of ROM be devoted to the modified FORTH kernel.

Still another system for altering software in embedded systems is disclosed in U.S. Patent Number 5,901,225, issued on May 4, 1999 to Ireton et al. The system includes an embedded system device coupled to an external memory. The device includes a non-alterable memory, including firmware, coupled to a processor. The device further includes a relatively small amount of patch RAM within the device also coupled to the processor. Patches are loaded from the external memory into the patch RAM. The device further includes a means for determining if one or more patches are to be applied. If the device detects a patch to be applied, the system loads the patch from the external memory into the patch RAM. The device also includes a breakpoint register. When the value of the program counter of the processor equals the value in the breakpoint register, a patch insertion occurs, i.e., the processor deviates from executing firmware to executing patch

instructions. The system described by Ireton et al. is quite complex.

SUMMARY OF THE INVENTION

It is therefore an object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory.

It is also an object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory which make efficient use of RAM.

It is another object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory which are relatively simple in architecture.

It is also an object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory which do not require any decision making elements in ROM to support future code modifications.

It is another object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory

which do not require any processor specific or processor dependent elements.

It is still another object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory which are applicable to any programming language.

It is also an object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory which do not limit the scope of future code modifications.

It is another object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory which incur minimum processing overhead.

It is still another object of the invention to provide methods and apparatus for modifying firmware code stored in a read only memory which minimize the size of ROM code segments needed to be replaced to fix an error.

In accord with these objects which will be discussed in detail below, the apparatus of the invention includes an embedded device having program code stored in ROM and an on-board or external RAM for storing modified code segments. The methods of the invention include structuring the ROM-based firmware so that a

RAM-based function is called prior to each potentially modifiable code segment. Prior to modifying the firmware, a dummy function is stored in RAM so that every call to RAM is simply returned to ROM. When a segment of code is to be modified, a replacement is stored in RAM and indexed by the return address of the function call. The system of the present invention is efficient as it uses very little RAM. It does not require any ROM-based decision making elements; and it is not limited to a particular programming language or processor. The system of the invention is most suitable for use in a computer peripheral which communicates with a higher level controller, e.g. a personal computer, from which replacement code can be downloaded. Alternatively, replacement code in RAM can be loaded by a small bootstrap program (i.e. a run-time system initialization program) stored in replaceable external ROM.

Additional objects and advantages of the invention will become apparent to those skilled in the art upon reference to the detailed description taken in conjunction with the provided figures.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a simplified block diagram of an embedded system according to the invention;

Figure 2 is a schematic illustration of the organization of ROM-based code according to the invention;

Figure 3 is a schematic illustration of the dummy function in RAM according to the invention;

Figure 4 is a schematic illustration of the replacement code in RAM indexed to return address; and

Figure 5 is a simplified flowchart of the operation of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Referring now to Figure 1, an embedded system 10 according to the invention includes a processor 12 which is coupled to a firmware ROM 14 and a RAM 16. As illustrated in Figure 1, the RAM 16 is ideally much smaller than the ROM 14, for example approximately one one hundredth the size of the ROM.

Turning now to Figure 2, according to the invention, the firmware stored in the ROM 14 is divided into code segments, e.g. 18a-18d, 20a-20d, 22a-22d, and 24a-24d. Prior to the start of each code segment, a call function with a (calling and) return

address is placed, e.g. at 18, 20, 22, and 24. The locations of the call functions are preferably organized for the most efficient use of RAM. It might seem practical to locate each call function at each logical break in the code, e.g. at the start of each new routine. However, in the case of very long routines, it can be more efficient to place call functions equally spaced throughout the routine. This will be better understood after considering the code listings below. Essentially, whenever replacement code is used, all of the code from the call function until the corrected code is reached is replaced. Thus if there are one thousand lines of code between call functions and only the six hundredth line of that code needs to be changed, six hundred lines will be replaced nonetheless. If the call functions were spaced every one hundred lines throughout the thousand line routine, no more than one hundred lines would need to be replaced. Thus, a balance must be struck between the number of call functions placed throughout the ROM-based code and the amount of RAM needed to make code replacements.

Figure 3 illustrates the structure of RAM 16 when none of the code segments in ROM is to be replaced. A dummy function 26, including instruction lines 26a-26d resides in RAM 16. Whenever a call function (18, 20, 22, 24 in Figure 2) is executed, the dummy function 26 responds by returning program execution to the return

address (which is usually the next line after the calling function).

Figure 4 illustrates the structure of RAM 16 when some of the code segments in ROM are to be replaced. In this case, the Return in the Dummy Function is replaced with a branch to the corresponding Patch Center Function. According to the presently preferred embodiment, a lookup table 28, including a "patch center function" (line 1 of the RAM table) provides the functionality necessary to determine whether a code segment has a replacement in RAM. Operation of the patch center function is described in more detail below with reference to Figure 5 and Code Listing 2. The lookup table is illustrated schematically at lines 2-4 of the RAM table, line 2 being the start of table (SOT) and line 4 being the end of table (EOT). Each entry in the lookup table refers to a return address (RA) and a patch address (PA). For example, the first entry refers to a return address of X+1 and a patch address of 5. This signifies that the code being replaced is a replacement (shown at 30 in Figure 4) for code segment X; the replacement code begins at line 5 in the RAM table and upon execution of the replacement code, execution continues at an address specified in the replacement code, typically the next segment after X. It is possible to replace only a portion of a code segment and return to a line within the original code segment to continue execution. As will be seen below in Code Listing 2, it is necessary to replace a contiguous set of code lines from the

call function forward. Similar entries are indicated for code segment Y and Z replacements which are illustrated at 32 and 34 in Figure 4.

Referring now to Figure 5, when the patch center function is called at 100, the lookup table entries are scanned at 102. If a return address match is found at 104, program execution is directed to the patch address at 106. The replacement code is executed at 108, and program execution is returned to a ROM address specified in the patch code at 110. So long as the end of the table has not been reached as determined at 112, the patch center function looks for a table entry for replacement code. When it is determined that the end of the table has been reached, program execution returns to the return address in ROM at 114. As mentioned above, if there is no replacement code in RAM, program execution returns to the return address specified by the call function which also acts as an index to the lookup table. If replacement code is executed, the return address is specified by the replacement code.

The operation of the invention will be better understood with reference to the following Code Listing 1 and Code Listing 2 which illustrate the structure of the code in ROM and RAM respectively.

```
 2  *******************************
 3  * Code Listing 1 (ROM Segment)
 4  *******************************
 5
 6  Code_in Rom:
 7           Patch_Call1()              ; Function call
 8           .
 9           .
10           .
11           Patch_Call1()
12           .
13           .
14           .
15           Patch_Call2()
16  ROM_Return_Addr2:                   ; Return Address (RA)
17           I = I + 10                 ; Segment code starts
18           K + K - 55                 ; Need modification !
19  ROM_Address2:
20           M = M + 88
21           .
22           .                          ; segment code ends
23           Patch_Call2()
24           J = J - 15
25           .
26           .
27           Patch_Call2()
28           .
29           .
30           .
31           Patch_Call3()
32           .
33           .
34           .
35           Patch_Call3()
```

<u>Code Listing 1 (ROM Segment)</u>

Referring to Code Listing 1, a code segment is shown with seven patch calls at lines 7, 11, 15, 23, 27, 31, and 35. Most of the code is not shown but is represented by dots, i.e. at lines 8-10, 12-14, 21, 22, 25, 26, 28-30, and 32-34. The code which will be modified follows patch call2() at line 15.

```
1  ********************************
2  * Code Listing 2 (RAM Segment)
3  ********************************
4  Code_in_RAM:
5
6  Patch_Call1                                  ; Dummy function
7            Return                             ; Return to ROM Code
8
9  Patch_Call2:                                 ; Dummy Function
10           Go to Patch_Center 2               ; Replace Return here !
11
12 Patch_Call3:                                 ; Dummy Function
13           Return                             ; Return to ROM Code
14           .
15           .
16           .
17
18 Patch_Center2:                               ; Patch Center Function
19
20           Save_Context()                     ; Save necessary registers
21
22           Address_Match = Patch_Address_Table_Search()
23                                              ; Lookup table searching
24
25           if (Address_Match == YES)
26                 Go to Patch_Code2            ; Have replacement code
27           else
28                 Restore_Context()            ; Restore the registers
29                 Return                       ; Return to ROM code
30
31 Patch_Center2_Lookup_SOT:                    ; Start of Lookup Table
32           ROM_Return_Addr2                   ; Return Address (RA)
33           Patch_Code2                        ; Patch Address (PA)
34           .
35           .
36           .
37 Patch_Center_Lookup_EOT:                     ; End of Lookup Table
38
39 Patch_Code2:                                 ; Patch Address (PA)
40           I = I + 10                         ; Replacement code
41           K = K - 99                         ; Replace K = K - 55 in ROM
42           Restore_Context()                  ; Restore the registers
43           Return to ROM_Address2
```

Code Listing 2 (RAM Segment)

Turning now to Code Listing 2, lines 6-13 represent the dummy functions. As shown, patch calls 1 and 3 are dummy functions which return the program to ROM. Patch call 2 directs the program

to patch center 2 which begins at line 18. Patch center 2 saves the context and checks the address match. The address match lookup table is illustrated at lines 31-37. If the address matches, the program is directed to patch code 2 which starts at line 39. If the address does not match, context is restored and the program is returned to ROM. As shown at lines 39-43, patch code 2 is designed to replace two lines of code in code listing 1, i.e. replace lines 17 and 18 of code listing 1 with lines 34 and 35 of code listing 2. After executing lines 40 and 41 of code listing 2, context is restored at line 42 and the program is returned to ROM at 43. The return address is indicated in code listings 1 and 2 as ROM_Address2. The actual return address is derived from knowledge of the code in ROM. It should be noted that line 40 of code listing 2 is identical to line 17 of code listing 1. Nevertheless, it is replaced because, as mentioned above, the function calls according to the invention allow and require replacement of all code from the function call through the corrected code.

There have been described and illustrated herein methods and apparatus for modifying program code stored in ROM. While particular embodiments of the invention have been described, it is not intended that the invention be limited thereto, as it is intended that the invention be as broad in scope as the art will allow and that the specification be read likewise. It will

therefore be appreciated by those skilled in the art that yet other modifications could be made to the provided invention without deviating from its spirit and scope as so claimed.